Leaf Classification:
4 Model Analysis (DT, RF, ANN, SVM)

BITS F464 (Machine Learning) Sem 2 of 2019-20
Final Comprehensive Assignment

NIKHIL SREEKUMAR NAIR (2016A7PS0049H)

This report compiles the implementation and performance details of four **classification** models that were trained on the ImageClef 2012 plant leaves dataset. The four models examined are **decision trees**, **random forests**, **neural networks** (multi-layer perceptron), and **support vector machines** (SVMs).

The dataset used here consists of extracted features amounting to **5163 leaf samples** spread across **122 target classes**. The full list of extracted features comes up to a total of **129 features**. One significant observation on initial inspection of the dataset is the **imbalance** in the members representing each class, with many classes as low as 2 examples, and one even amounting to just 1 example. This has been dealt with (or ignored conveniently) differently wrt each of the four models.

Owing to the imbalance in the dataset, all the **performance metrics** discussed will be only **macro-averages** to give representation to all classes equally.

1 Decision Tree Classifier

1.1 Implementation details

The decision tree classifier was implemented using **sklearn.tree.DecisionTreeClassifier()**, while major data manipulation was achieved with the help of **pandas** and **numpy**. Grid search cross validation to tune hyperparameters was enabled through **sklearn.model_selection.GridSearchCV()** and performance metrics were reported using **sklearn.metrics**.

For the data split, the example that was the sole representative of the singleton class mentioned earlier was held out initially. A **stratified split** into train and test sets (0.7 : 0.3) was executed, after which the held out example was included with the train set.

1.2 Dry run and Feature Selection

In order to work with a **baseline**, a **dry run** of the classifier was executed without the removal of any features. This run achieved an **accuracy score of 0.6985**.

A **feature importance evaluation** (Fig 1) of all the 129 features was examined, post which all the features were sorted in the decreasing order of their importance. Importance here refers to how important the tree deems a split at that particular feature, based on post-split information gain.

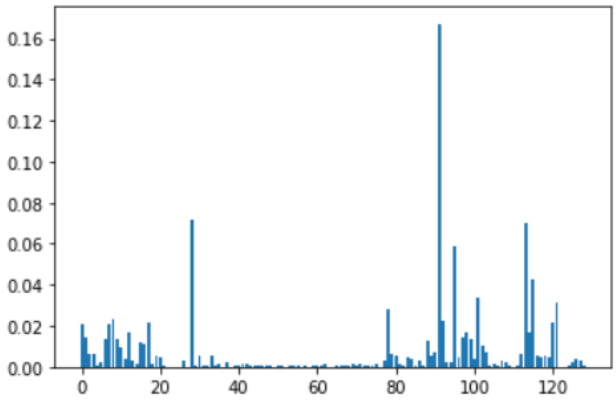


Fig 1: (x: feature numbers, y: feature importance measure)

An approximate split was decided at shortlisting the **top 90% important features** (Fig2), owing to observation of a peak accuracy at the same. This filtered dataset was then used for the subsequent steps.

```
shortlisting top 75.0 % features...
decision tree classifier: 0.7075532601678503
..............
shortlisting top 80.0 % features...
decision tree classifier: 0.7043253712072305
..............
shortlisting top 85.0 % features...
decision tree classifier: 0.7049709489993544
..............
shortlisting top 90.0 % features...
decision tree classifier: 0.7204648160103292
..............
shortlisting top 95.0 % features...
decision tree classifier: 0.7017430600387347
..............
shortlisting top 100.0 % features...
decision tree classifier: 0.6959328599096191
..............
```

Fig 2: resultant accuracies post removal of x% least important features

1.3 Hyperparameter tuning

Hyperparameters tuned were primarily motivated towards **controlling overfitting** possibilities due to fully growing the tree to max depth by finishing at zero entropy leaves. Hyperparameters tuned were primarily:

max_features: whether to consider all the features before deciding on a split. Primarily used to contemplate a trade-off between execution time and performance.

min_samples_split: an indirect hyperparameter that checks the depth (**height**) growth of the tree, and thus overfitting possibilities. This parameter decides what the minimum count of examples at a node is for the tree to be able to branch further from there.

min_samples_leaf: another hyperparameter to check overfitting. This value tells the tree that it need not have zero entropy at the leaves, to prevent unnecessarily **wide-spread** trees, and to increase generality to an extent.

```
# grid search for decision tree
s_plit = [x for x in range(2,11)]
l_eaf = [x for x in range(1,11)]

params = {'max_features': [None, 'auto', 'sqrt', 'log2'],
          'criterion':["entropy"],
          'min_samples_split':s_plit, # controls overfitting, also depth of tree
          'min_samples_leaf':l_eaf, # controls overfitting, leaves need not have zero entropy
          'random_state':[23]}

X_train_tree, y_train_tree, X_test_tree, y_test_tree = util_vanilla(df_tree)

grid_tree = GridSearchCV(tree.DecisionTreeClassifier(), param_grid=params, verbose=3, n_jobs=-1)
grid_tree.fit(X_train_tree, y_train_tree)
```

```
Fitting 5 folds for each of 360 candidates, totalling 1800 fits
```

Fig 3: Grid search cross-validation to tune hyperparameters

Tuning and shortlisting of the hyperparameters was achieved using a **grid search** across **360 possible combinations**. These were validated within the training set itself, using **5-fold cross validation**. The best parameters were finalised at the default representation of the classifier, however with

max_features = None (default, i.e. all features were considered),

min_samples_leaf = 1, and

min_samples_split = 2

1.4 Performance Metrics

Naturally, as a result the train set did achieve an accuracy of 1.0. The test set was able to secure the following performance scores, as seen in Table 1 (precision, recall, and f-1 score reported are all **macro-averages**, and not weighted).

Metric	Accuracy	Precision	Recall	F-1 Score
Score	0.7205	0.62	0.63	0.61

Table 1: decision tree performance metrics

2 Random Forest Classifier

2.1 Implementation details

The decision trees ensemble, the random forest classifier was implemented using **sklearn.ensemble.RandomForestClassifier()** with **pandas, numpy, sklearn.metrics,** and

sklearn.model_selection.GridSearchCV() being used to fulfil the same roles as seen earlier in decision trees.

Again, for the data split, the example that was the sole representative of the singleton class mentioned earlier was held out initially. A **stratified split** into train and test sets (0.7 : 0.3) was executed, after which the held out example was included with the train set.

2.2 Dry run and Feature Selection

In order to work with a **baseline**, a **dry run** of the classifier was executed without the removal of any features. This run achieved an **accuracy score of 0.9219**.

A similar **feature importance evaluation** (Fig 4) of all the 129 features was examined wrt the random forest classifier this time, post which all the features were sorted in the decreasing order of their importance.

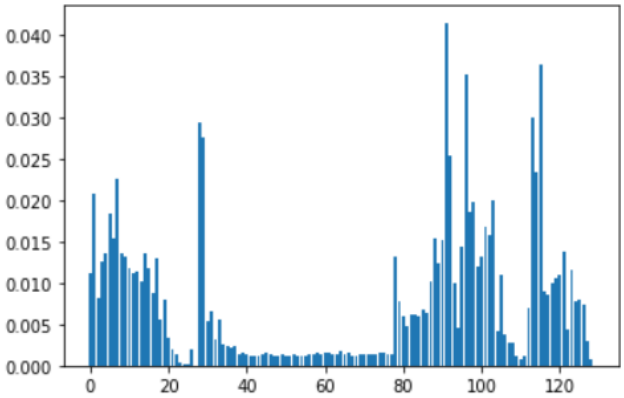


Fig 4: (x: feature numbers, y: feature importance measure)

An approximate split was decided at shortlisting the **top 85% important** features (Fig 5), owing to observation of a peak accuracy of 0.9380 at the same. This filtered dataset was then used for the subsequent steps.

```
shortlisting top 75.0 % features...
random forest classifier: 0.9160748870238864
..............
shortlisting top 80.0 % features...
random forest classifier: 0.9302775984506133
..............
shortlisting top 85.0 % features...
random forest classifier: 0.9380245319561007
..............
shortlisting top 90.0 % features...
random forest classifier: 0.9367333763718528
..............
shortlisting top 95.0 % features...
random forest classifier: 0.9309231762427372
..............
shortlisting top 100.0 % features...
random forest classifier: 0.9173660426081343
..............
```

Fig 5: resultant accuracies post removal of x% least important features

2.3 Hyperparameter tuning

Hyperparameters tuned were primarily motivated towards controlling overfitting possibilities due to fully growing individual trees to max depth by finishing at zero entropy leaves. Hyperparameters tuned were primarily:

n_estimators: the number of trees the ensemble classifier considers (constructs) and thereafter combines.

min_samples_split and **min_samples_leaf**, analogous to individual trees (as seen in the Decision Trees illustration)

```
# grid search for random forest
params = {'criterion':["entropy"],
 'n_estimators':[90,100,110,120], # how many trees to consider
 'min_samples_split':[2, 3, 4, 5, 6, 7, 8, 9], # every indiv tree: controls overfitting, also depth of tree
 'min_samples_leaf':[1, 2, 3, 4, 5], # every indiv tree: controls overfitting, leaves need not have zero entropy
 'random_state':[23],
 'n_jobs':[-1]}

X_train_rf, y_train_rf, X_test_rf, y_test_rf = util_vanilla(df_rf)

grid_rf = GridSearchCV(RandomForestClassifier(), param_grid=params, cv=3, n_jobs=-1, verbose=3)
grid_rf.fit(X_train_rf, y_train_rf)
```

```
Fitting 3 folds for each of 160 candidates, totalling 480 fits
```

Fig 6: Grid search cross-validation to tune hyperparameters

Tuning and shortlisting of the hyperparameters was achieved using a **grid search** across **160 possible combinations**. These were validated within the training set itself, using **3-fold cross validation**. The best parameters finalised were as follows.

n_estimators = 110,

min_samples_leaf = 1, and

min_samples_split = 2

2.4 Performance Metrics

The train set achieved an accuracy of 1.0. The test set was able to secure the following performance scores, as seen in Table 1 (precision, recall, and f-1 score reported are all **macro-averages**, and not weighted).

Metric	Accuracy	Precision	Recall	F-1 Score
Score	0.9374	0.91	0.88	0.89

Table 2: random forest performance metrics

3 Artificial neural networks

3.1 Implementation Details

Majority of the neural network related requirements (layers, model architecture, regularization, dropout, activations) were implemented using the **Keras** neural network library. Reprising earlier roles were libraries **pandas**, **numpy**, and **sklearn.metrics**. Also, **sklearn.preprocessing** was used for one-hot encoding the target variable and also for normalising the data.

3.2 Pre-processing

Owing to gross disparities in the feature ranges, a **normalisation of all the feature** values was undertaken. Also, owing to the problem representing a multi-classification implementation, the **target output variable was one-hot encoded** to fit in with the **softmax** activation that would be applied as activation in the final layer of both models. Post this, the aggregate dataset was split into **train**, **validation**, and **test** sets (0.6 : 0.2 : 0.2). The validation set was used to evaluate overfitting while teaching the model with the train set.

3.3 Motivation for choice of hidden layers

The general ideas for number of neurons and number of hidden layers were a result of a series of trial and error exercises. From the limited exposure of trials that were conducted, a **3-layer** model with nodes distributed as **129/256/122** and a **4-layer** model with nodes **129/512/256/122** were shortlisted. Also, **ReLu** as an activation for every layer (except target) was deemed the most suitable after running a few trials against sigmoid. The target layer naturally had a **softmax** activation.

Initially, intermediate values between 129 and 122 were considered for hidden layer counts. Despite this, larger values were giving better results from observable raw runs. This could arguably be because of inconvenient non-deterministic results; irrespective, these two aforementioned models are discussed in slight detail, wrt their implementation, design, and performance scores.

3.4 Hyperparameters – network, regularization, dropout

Conventionally, any multilayer neural network has the hyperparameters **learning rate**, **batch size**, and **epochs**. However, in this implementation, learning rate as a tuneable parameter is forgone on account of using the Adam Optimisation in place of Stochastic Gradient Descent. This choice was made to practice better performance trade-offs, as well as to exploit Adam's ability to self-adjust the learning rate through methods of momentum and moving averages.

Also, the loss function used was the **categorical cross-entropy** (i.e $-\sum y_{o,c} \cdot \log(p_{o,c})$, where c ranges across all M classes, y is the binary indicator true or false of the example wrt that class, and p is the predicted classification) loss function.

With this, the major hyperparameters that were tuned and examined are as follows.

batch size: tuned in powers of 2, specifically 32, 64, 128, 256. Got the best raw performances from models with batch size 128 for both the models.

epochs: tuned in multiples of 100, specifically 100, 200, 300, 400, 500, 600 as baseline starters. Observed 500 and 300 as optimum epoch durations for the three and four-layer models respectively. This is primarily owing to the fact that the deeper model was able to approximate required the functions within fewer runs.

regularization constant: L2 regularization was found to be the more suitable in both examples. Moreover, a constant value of 0.01 across various iterations was determined to be enough (among 1, 0.1, 0.01, 0.001).

dropout ratios: for both the three and four- layer networks, dropout rates were toggled among 0.25, 0.5, 0.6 and 1.0 in different combinations.

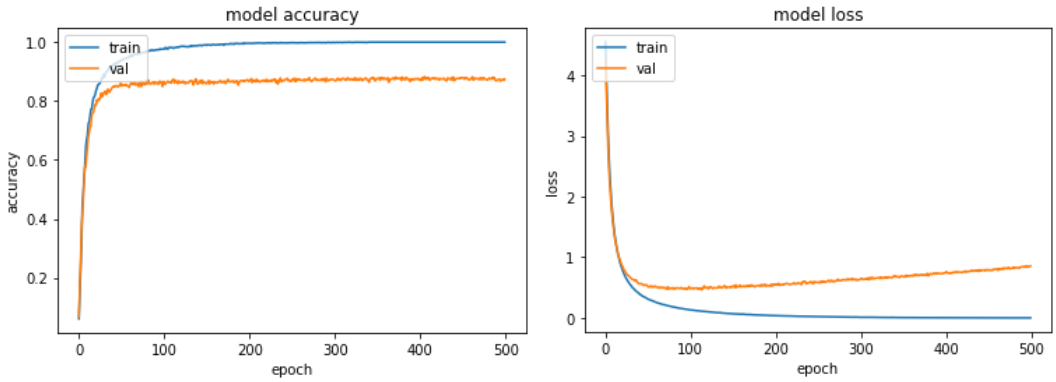
3.5 129/256/122 network model

3.5.1 Major design developments

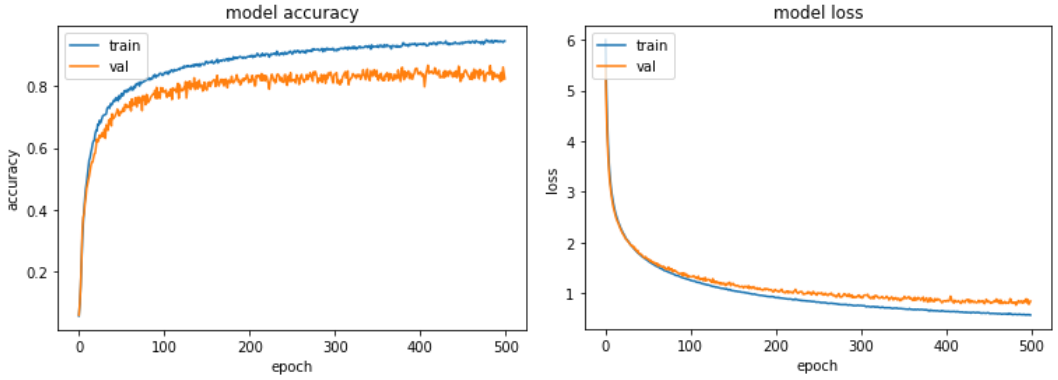
- Training iterations started at baseline of 500 epochs. Straight off the bat, the model was terribly overfit.
- In order to restrict the overfitting, certain degree of generality was introduced with lambda set at 0.01. Overfitting did decrease to a certain observable degree. Consequent incremental measures were introduced to completely do away with overfitting.
- Dropouts in different combinations were trialed. On observable improvement with dropout = 0.6, the epochs were increased to 800 to lengthen the model's learning duration.
- Finally, on account of comparable values between training and validation losses, model4 (see Section 3.5.2) was chosen as the final neural network.

3.5.2 Accuracy-Loss Plots

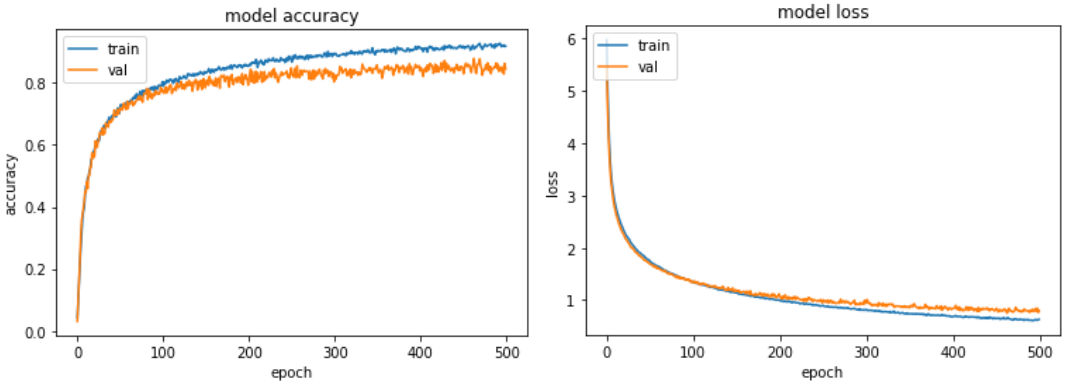
The following set of plots illustrate the major design developments in constructing and evaluating the three-layer network. These plots also illustrate the incremental improvement of the model's overfitting.



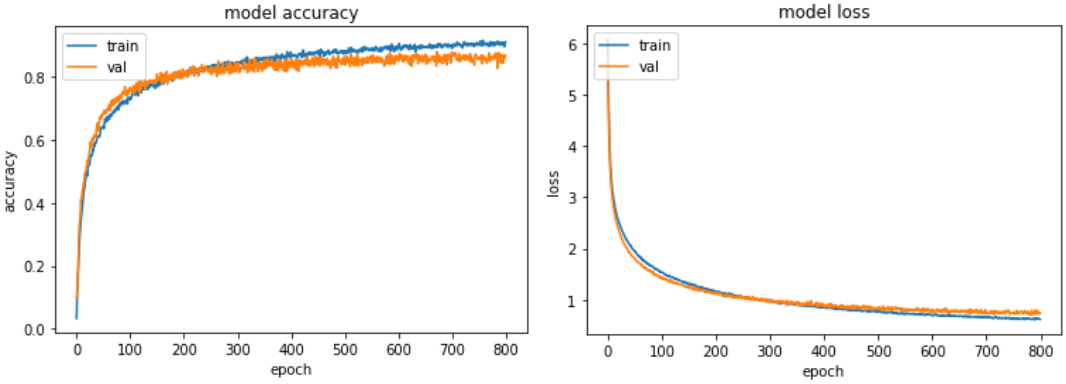
model0: ['l2', 'drop', 'epochs'] = [0.0, 1.0, 500]



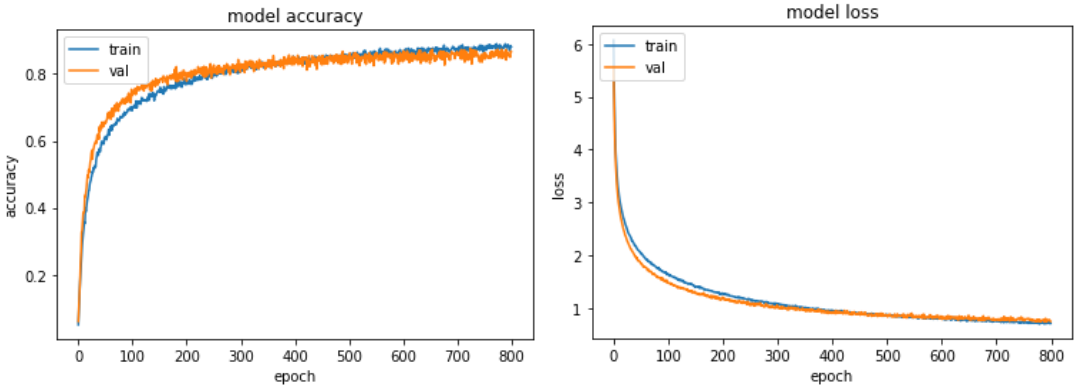
model1: ['l2', 'drop', 'epochs'] = [0.01, 1.0, 500]



model2: ['l2', 'drop', 'epochs'] = [0.01, 0.25, 500]



model3: ['l2', 'drop', 'epochs'] = [0.01, 0.5, 800]



model4: ['l2', 'drop', 'epochs'] = [0.01, 0.6, 800]

3.5.3 Performance Metrics

The model's (i.e. model4: ['l2', 'drop', 'epochs'] = [0.01, 0.6, 800]) testing accuracy and loss, along with the training and validation values for the same (i.e. what it ceased with the closing batch at 800 epochs):

Data Subset	Train	Validation	Test
Accuracy	0.8809	0.8664	0.8364
Loss	0.7078	0.7459	0.7962

Table 3: 3-layer train/validation/test accuracy and loss

In addition, the following are the performance metrics scores (accuracy, precision, recall, and f1 score) of the shortlisted model.

Metric	Accuracy	Precision	Recall	F-1 Score
Score	0.8364	0.7938	0.7914	0.7766

Table 4: 3-layer network performance metrics

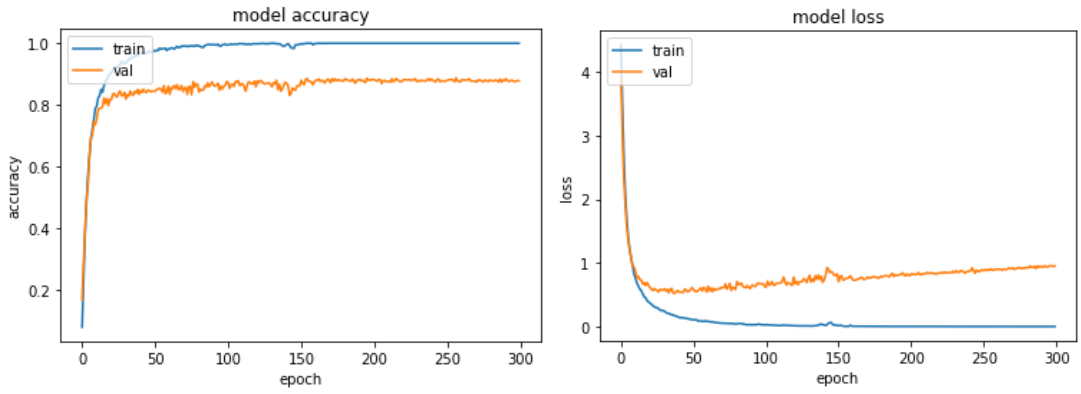
3.6 129/512/256/122 network model

3.6.1 Major design developments

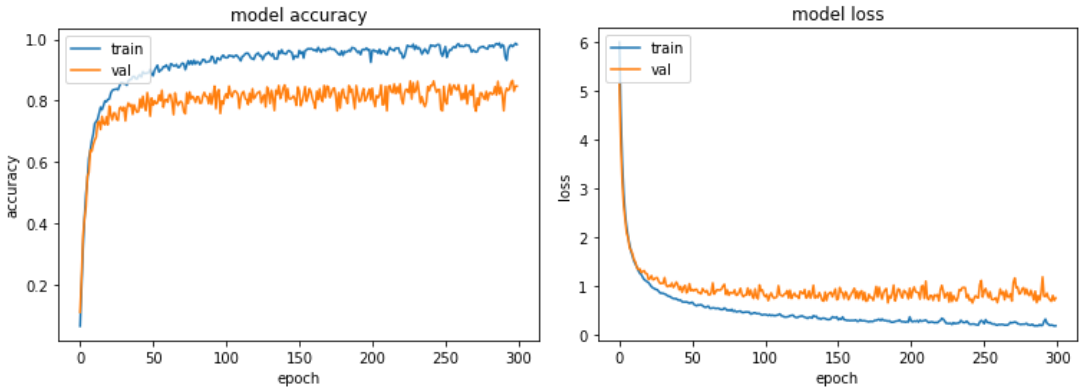
- Training iterations started at baseline of 300 epochs. Again, in order to restrict the overfitting, certain degree of generality was introduced with lambda set at 0.01.
- Dropouts in different combinations were trialed. A tangible improvement with dropout ratios as 0.5 and 0.6 for the two hidden layers were found to be the best mix.
- Finally, on account of comparable values between training and validation losses, model4 (see Section 3.6.2) was chosen as the final neural network. No change to the number of epochs was deemed necessary.

3.6.2 Accuracy-Loss Plots

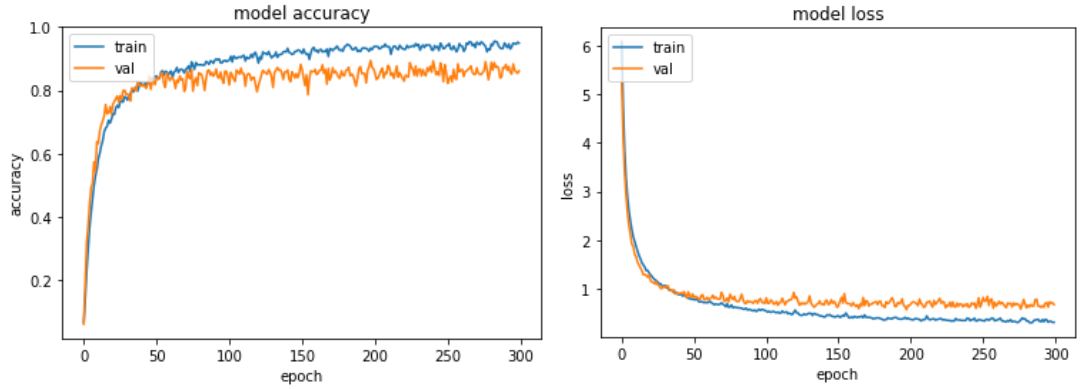
Analogous to section 3.5.2.



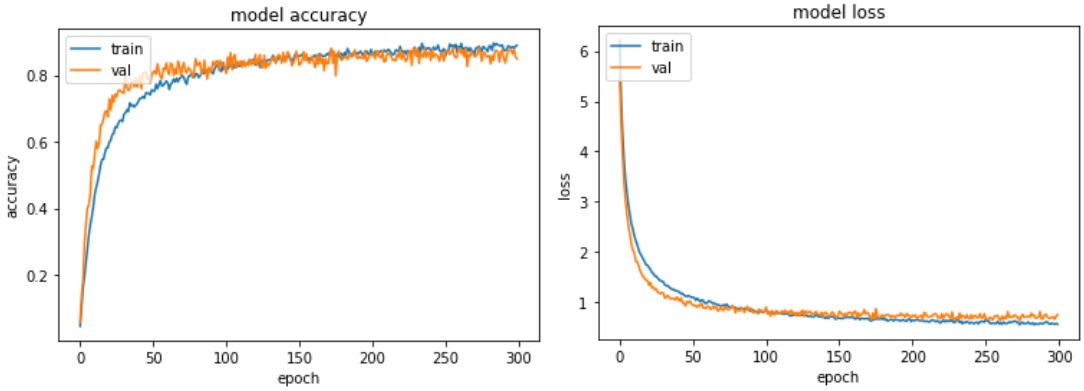
model0: ['l2', 'drop1', 'drop2', 'epochs'] = [0.0, 1.0, 1.0, 300]



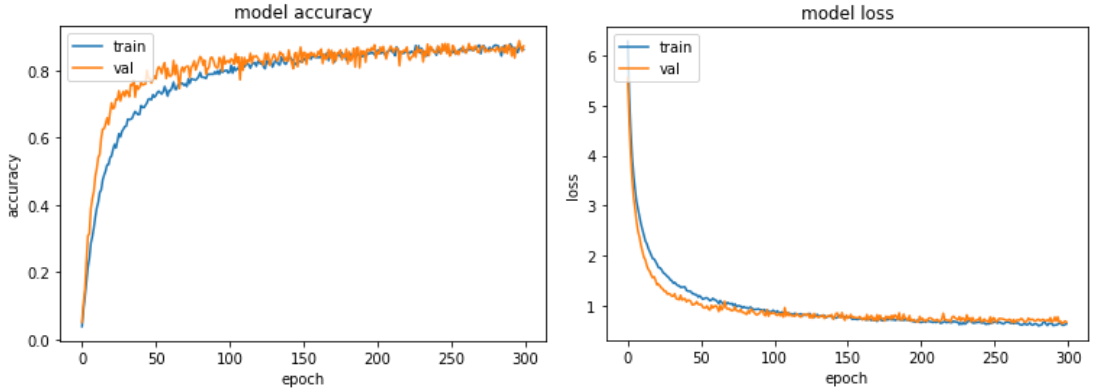
model1: ['l2', 'drop1', 'drop2', 'epochs'] = [0.01, 1.0, 1.0, 300]



model2: ['l2', 'drop1', 'drop2', 'epochs'] = [0.01, 0.25, 0.25, 300]



model3: ['l2', 'drop1', 'drop2', 'epochs'] = [0.01, 0.5, 0.5, 300]



model4: ['l2', 'drop1', 'drop2', 'epochs'] = [0.01, 0.5, 0.6, 300]

3.6.3 Performance Metrics

The model's (i.e. model4: ['l2', 'drop1', 'drop2', 'epochs'] = [0.01, 0.5, 0.6, 300]) testing accuracy and loss, along with the training and validation values for the same (i.e. what it ceased with the closing batch at 300 epochs):

Data Subset	Train	Validation	Test
Accuracy	0.8615	0.8712	0.8374
Loss	0.6487	0.6853	0.7772

Table 5: 4-layer train/validation/test accuracy and loss

In addition, the following are the performance metrics scores (accuracy, precision, recall, and f1 score) of the shortlisted model.

Metric	Accuracy	Precision	Recall	F-1 Score
Score	0.8374	0.7735	0.7774	0.7503

Table 6: 4-layer network performance metrics

4 Support Vector Machine

4.1 Implementation Details

The SVM classifier used in this assignment was an implementation of **sklearn.svm.SVC()**. Fulfilling roles from earlier examples include **pandas**, **numpy**, **sklearn.metrics**, and **sklearn.model_selection.GridSearchCV()**.

4.2 Pre-processing

Owing to gross disparities in the feature ranges, a **standardisation of all the feature** values was undertaken. Standardisation, and not normalisation was practiced owing to SVMs being models that take into account distance metrics (like when maximising margin). **Distance-based models** like SVMs and KNNs would therefore require standardisation over normalisation.

A **stratified split** into train and test sets (0.7 : 0.3) was executed, after which the held out example (from the singleton class) was included with the train set.

To illustrate the significance of the standardisation step, the following accuracy scores of dry runs of the raw data splits and the standardised splits give a good picture:

Kernel	Raw Data	Standardized Data
Linear	0.8302	0.9051
Polynomial (degree=3)	0.1117	0.6856
Radial Basis Function	0.1394	0.8760
Sigmoid	0.0471	0.7921

Table 7: raw vs standardised data accuracies with dry (baseline) runs

4.3 Hyperparameter Tuning

Hyperparameters tuned were primarily motivated towards both **improving model accuracy/precision** and also **controlling overfitting**. Hyperparameters tuned were primarily:

C: cost parameter (cost of misclassification), i.e. how much the SVM is permitted to "bend" with the data. Lower costs usually translate to smoother decision boundaries, whereas higher leads to more bending around the example points.

kernel: with choices among **linear**, **polynomial**, **radial basis function**, and **sigmoid.**

gamma: the tuning parameter for the radial basis function. It signifies how far the influence of a single training example reaches. Low gamma values translate to farther away influence (i.e. looser decision boundaries surrounding training examples), and the inverse for high gamma values.

The **degree** hyperparameter has been intentionally omitted as the default degree taken for the kernel = 'poly' parameter setting is 3, which is capable of adapting to both quadratic and cubic decision boundaries.

```
# GridSearchCV for SVM

param_grid = {'C': [0.1, 1, 10, 100],
              'gamma': [1, 0.1, 0.01, 0.001],
              'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
              'random_state':[23]}
grid = GridSearchCV(svm.SVC(),param_grid,refit=True,verbose=3)
grid.fit(X_train_scaled,y_train)
```

```
Fitting 5 folds for each of 64 candidates, totalling 320 fits
```

Fig 7: Grid search cross-validation to tune hyperparameters

Tuning and shortlisting of the hyperparameters was achieved using a **grid search** across **64 possible combinations**. These were validated within the training set itself, using **5-fold cross validation**. The best parameters were finalised as follows.

C = 100,

Kernel = radial basis function, and

Gamma = 0.001

4.4 Performance Metrics

The train set achieved an accuracy of 0.9986. The test set was able to secure the following performance scores, as seen in Table 1 (precision, recall, and f-1 score reported are all **macro-averages**, and not weighted).

Metric	Accuracy	Precision	Recall	F-1 Score
Score	0.9083	0.84	0.82	0.82

Table 8: random forest performance metrics

5 Conclusion

For this particular dataset, taking **accuracy alone does not** do the models any justice. In fact, owing to **class imbalances** that are clearly prevalent within the dataset, the precision and recall scores become more important in particular. The **F1 score** gives a nice score balancing the precision and recall scores, making that the preferred metric when dealing with an uneven class distribution.

Fig 8 gives a pictorial representation of all the models performances, on both metrics – the accuracy and the macro-averaged F1 scores.

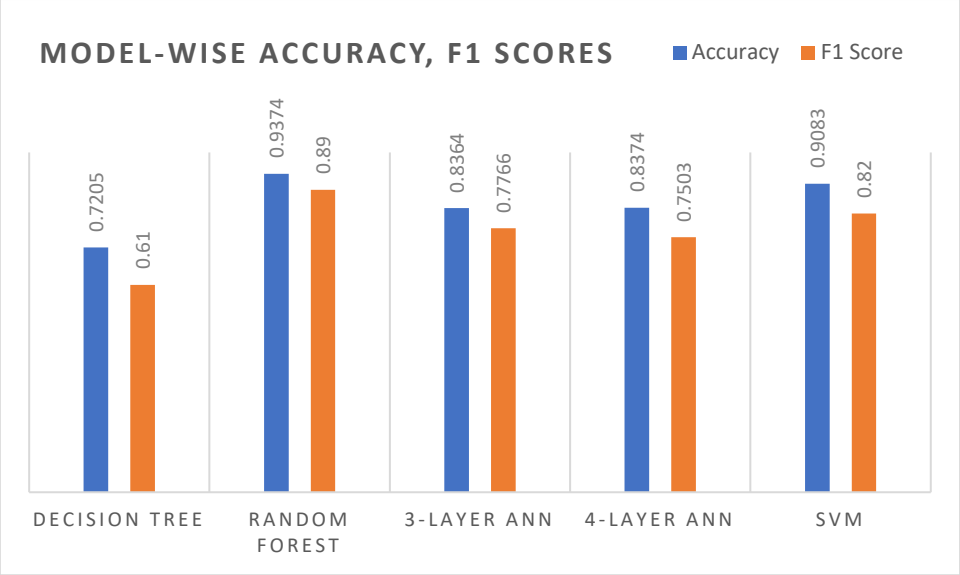


Fig 8: Accuracy and F1 scores compared across all models

Immediate observations and understandings that come to light here are:

- The **random forest classifier is the best model overall**, and scores better on not only the F1-score, but also the accuracy. The next best model out of the lot is the SVM.
- The random forest classifier's performance does not come as a surprise. Random forests are one of the **popular choices for highly unbalanced datasets** such as this, and have been known to give successful runs for the same. Moreover, they are **robust when it comes to overfitting**.
- Coming to SVMs, they are generally known to give good results with balanced datasets. However, with class-imbalanced datasets, they tend to give sub-optimal results. This is primarily attributed to the model drawing its decision boundaries primarily wrt to the heavier classes, tending to ignore the smaller classes in general.
- The **SVM's performance here can perhaps be a situation of statistical convenience** wrt this test set's spatial arrangement.
- For **decision trees**, the **most significant disadvantage is its tendency to easily overfit** any training it incurs.
- On the topic of **neural networks**, there might be better performing networks naturally. This exercise made use of only a very limited sample space of all the possibilities for a suitable implementation. However, when seen from a **time-vs-performance trade off**, the **random forest classifier** does take the win for this specific case.